HEAP

Heap

- A heap is a data structure used to implement an efficient priority queue.
- The idea is to make it efficient to extract the element with the highest priority the next item in the queue to be processed.

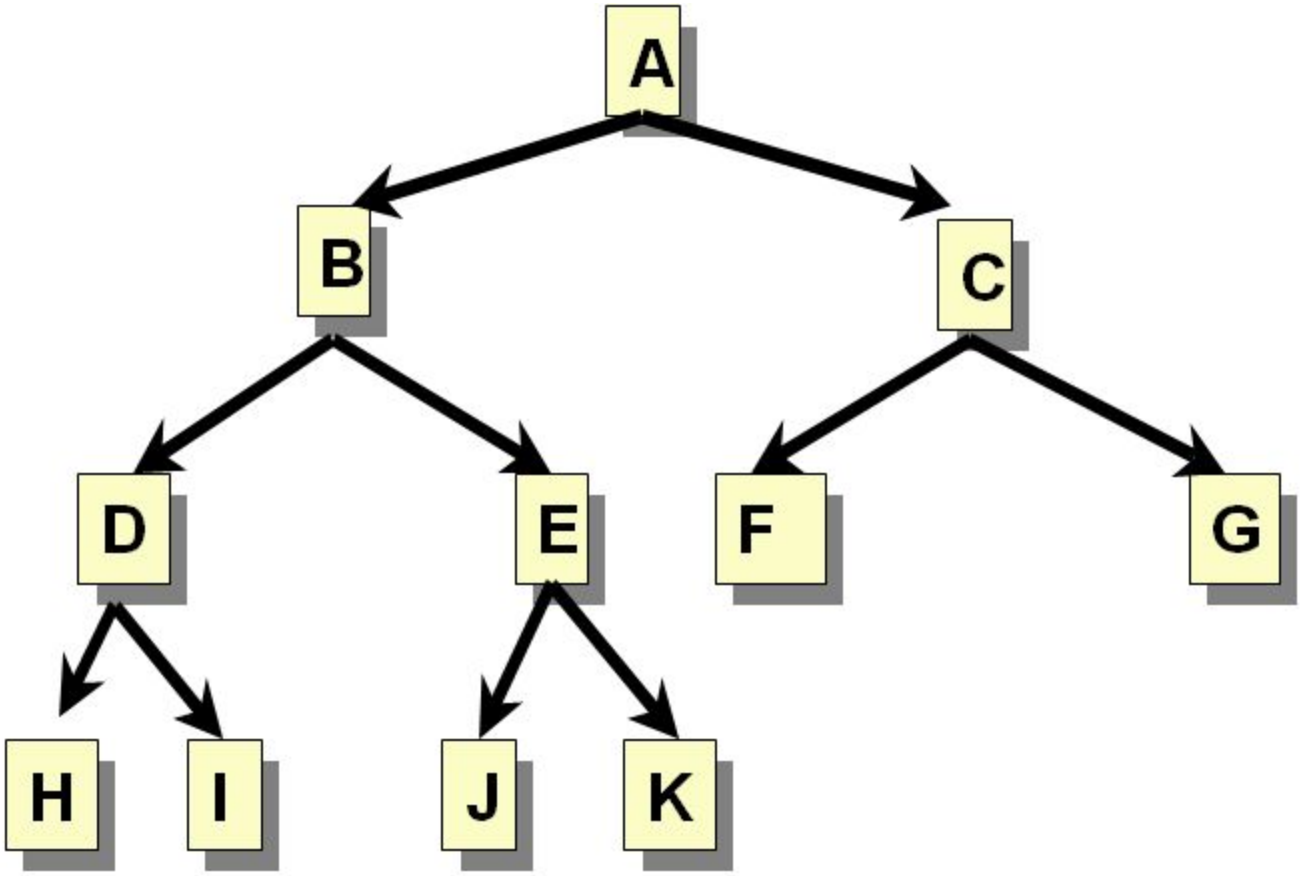
Heap

- Definition in Data Structure
  - Heap: A special form of complete binary tree such that key value of each node is no smaller (larger) than the key value of its children (if any).
- Max-Heap: root node has the largest key.
  - A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.
- Min-Heap: root node has the smallest key.
  - A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

Complete Binary Tree

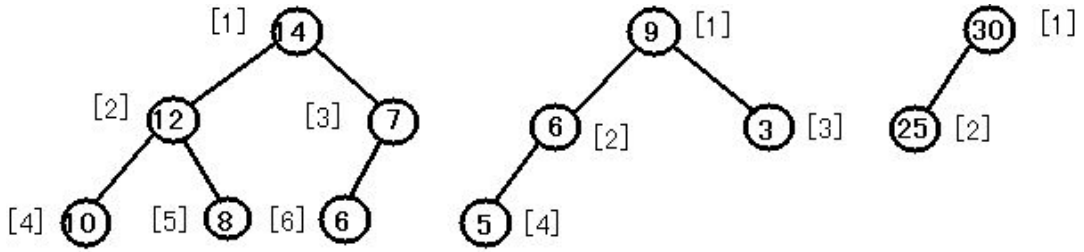
- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Complete Binary Trees - Example

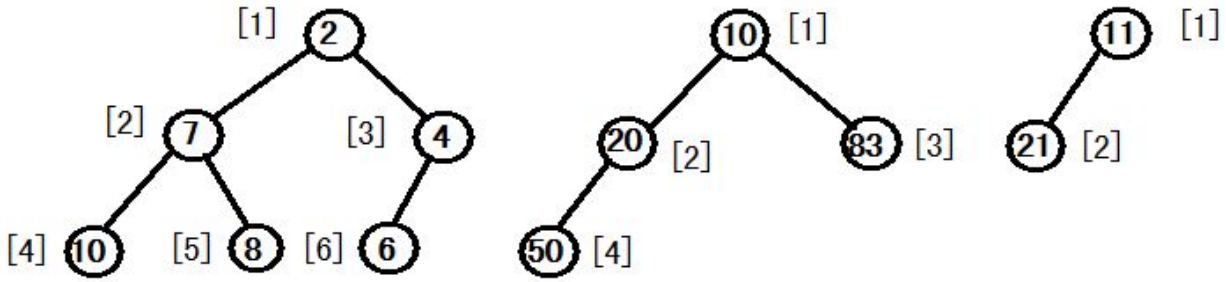


Heap

- Example:
  - Max-Heap



  - Min-Heap



ADT of Heap

- Notice:
  - Heap in data structure is a complete binary tree! (**Nice representation in Array**)
  - Heap in C++ program environment is an array of memory.
  - Stored using array in C++

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5

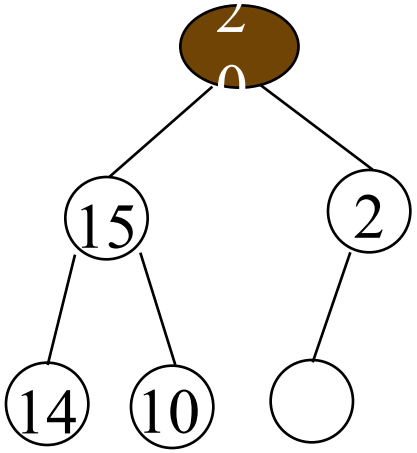
Heap

- Operations
  - Creation of an empty heap
  - Insertion of a new element into the heap
  - Deletion of the largest(smallest) element from the heap
- Heap is complete binary tree, can be represented by array. So the complexity of inserting any node or deleting the root node from Heap is O(height) = O( ). $\log_2 n$

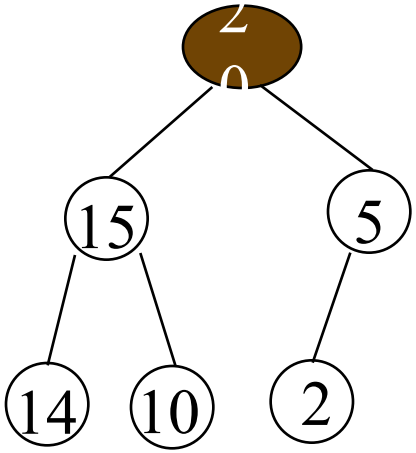
Heap

- Given the index i of a node
- Parent(i)
  - return i/2
- LeftChild(i)
  - return 2i
- RightChild(i)
  - Return 2i+1

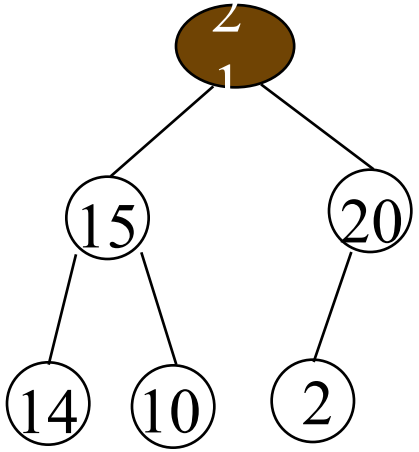
Example of Insertion to Max Heap



initial location of new node



insert 5 into heap



insert 21 into heap

Insertion into a Max Heap

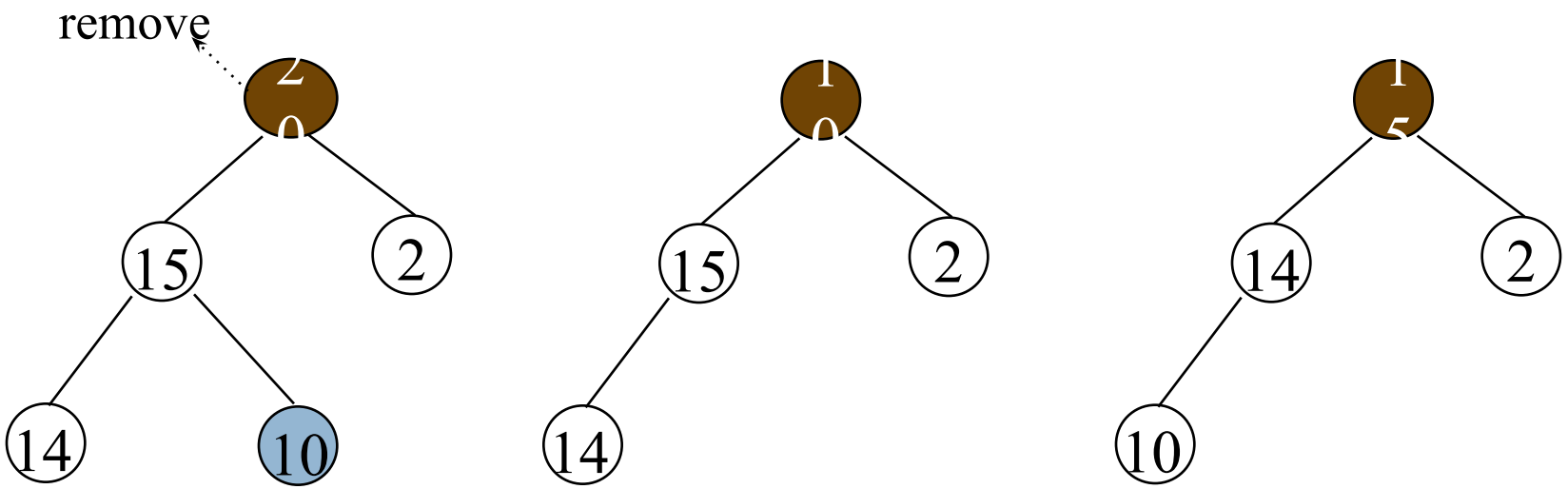
- To insert a node into the heap, a hole is first created at the next right position available within the bottom layer. If the bottom layer is full, a new layer is started from the left. If an array is used to implement the heap, a size check must first be performed to avoid array overflow.

- Values above a hole within the structure are bubbled down into the hole, so the hole "bubbles up" to the position where the hole can receive the value to be inserted while maintaining the heap ordering property.

Insertion into a Max Heap

```
void insert_max_heap(element &x)
{
if(n==MaxSize)
{
   Heapfull();
   Return;
}
n++;
int i=0;
for(i=n;i>=1;){
   if(i==1)
        break;
   if(x.key<=heap[i/2].key)
        break;

   Heap[i]=heap[i/2];
   i=i/2;
   }
Heap[i]=x;
}
```

Example of Deletion from Max Heap



Deletion From Max Heap

- Removal of the top node creates a hole at the top which is "bubbled" downwards by moving values below it upwards, until the hole is in a position where it can be replaced with the rightmost node from the bottom layer. This process restores the heap ordering property.

Deletion From Max Heap

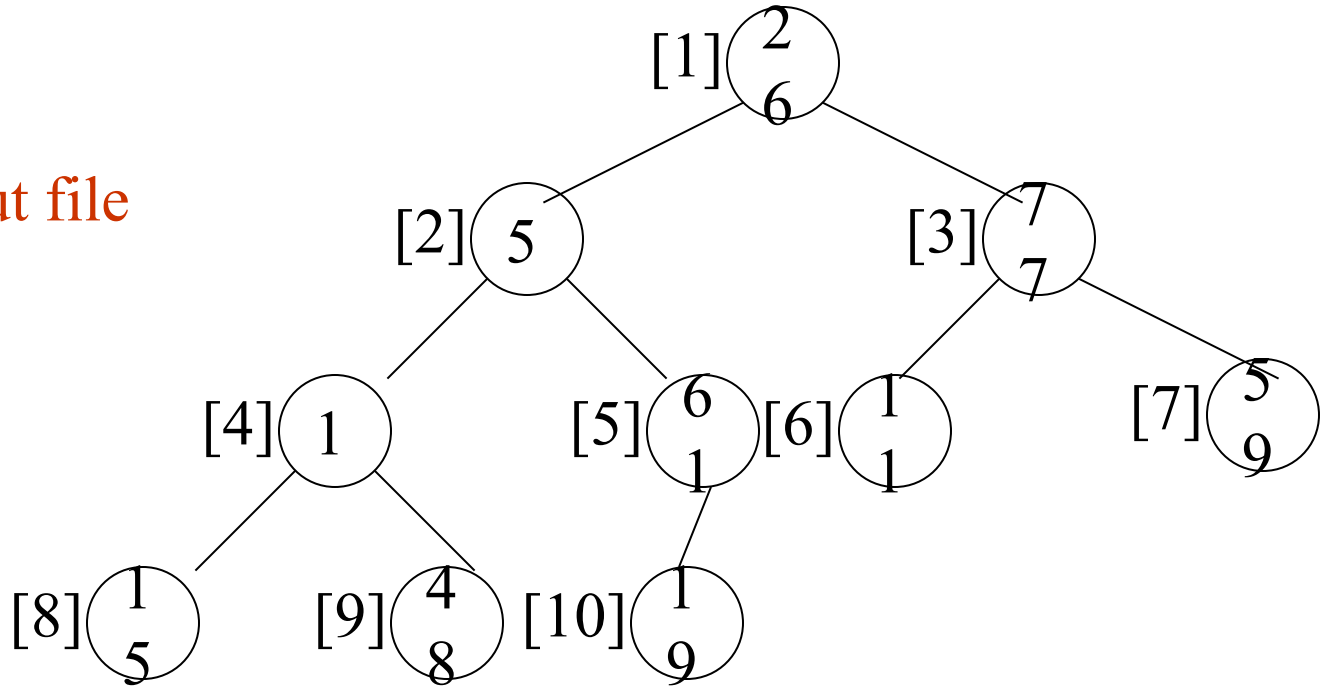
```
void deleteMax(Element &x){

if(!n){
     HeapEmpty();
     return;
}
x=Heap[1];
Element k=Heap[n];
n--;

for(int i=1, j=2; j<=n;){
     if(j<n)
          if(Heap[j].key<Heap[j+1].key)
          j++;
     if(k.key>=Heap[j].key)
          break;
     Heap[i]=Heap[j];
     i=j;
     j=j*2;
}
Heap[i]=k;
}
```

Application On Sorting: Heap Sort

- See an illustration first
  - Array interpreted as a binary tree

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

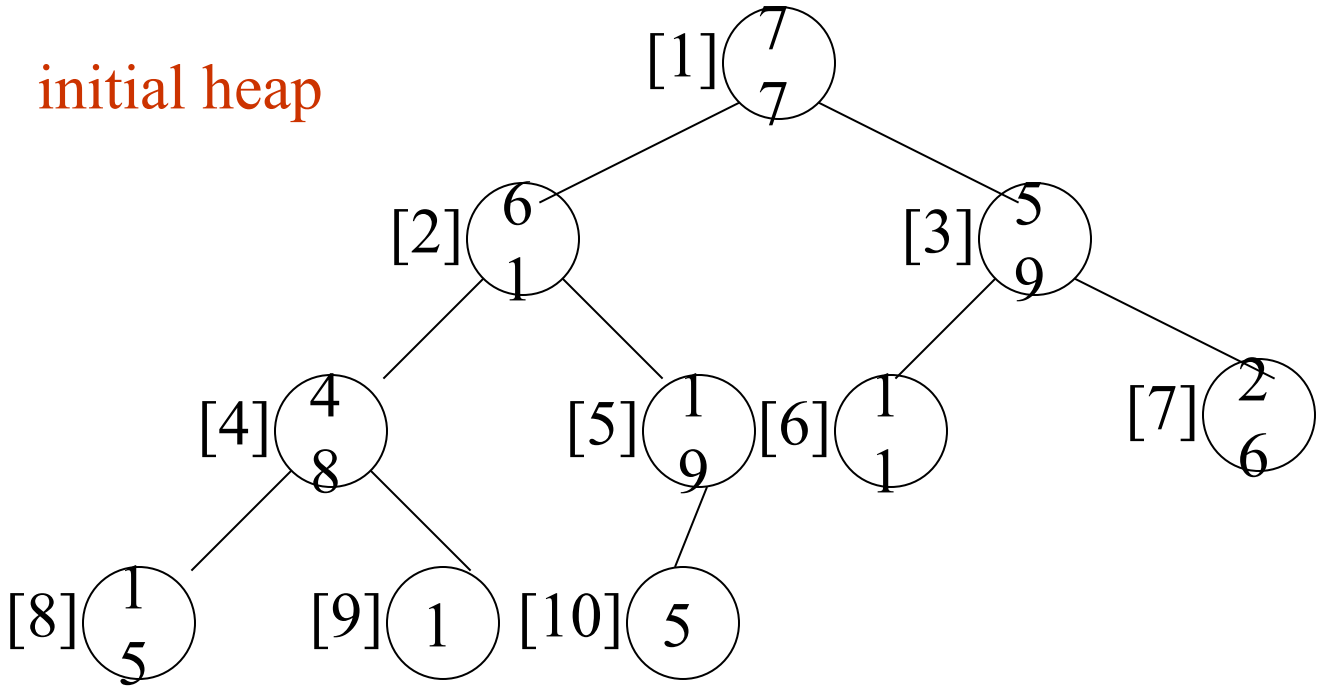
input file



Heap Sort

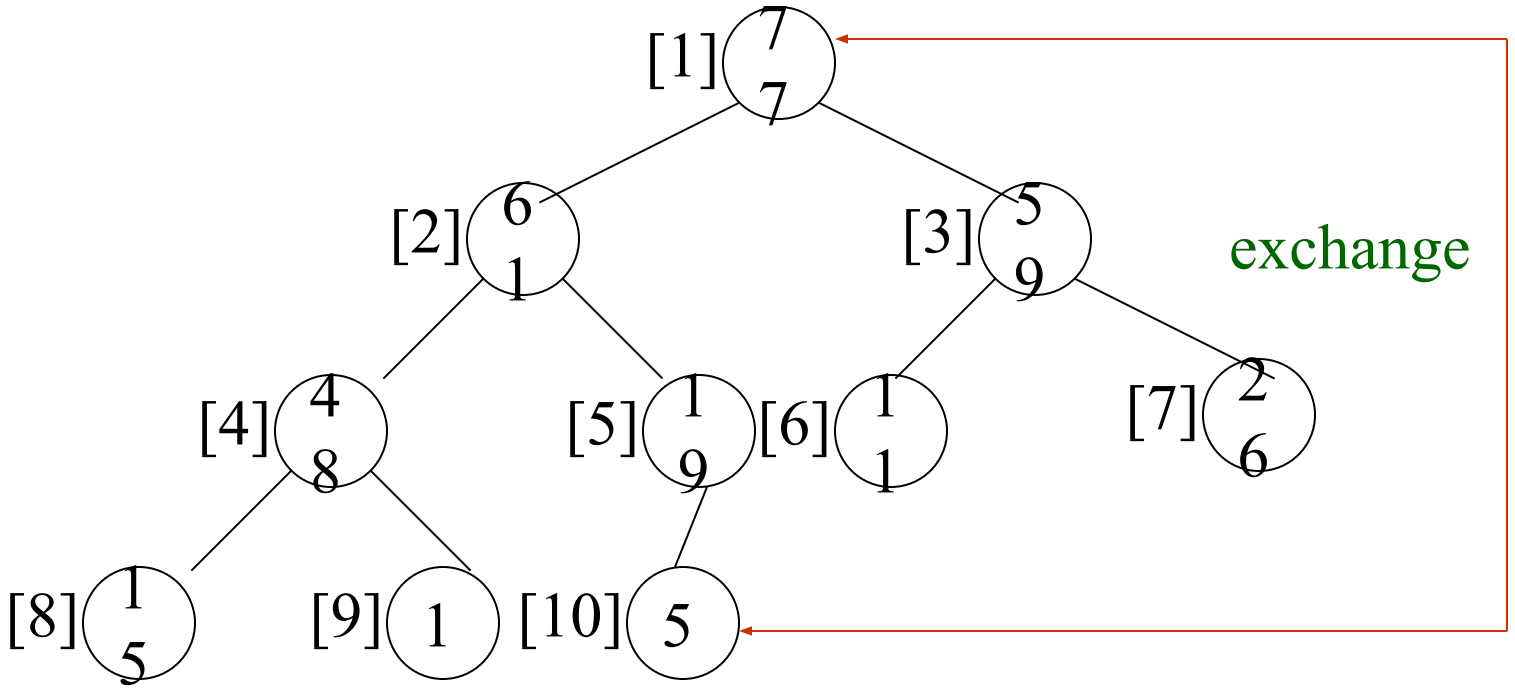
- Adjust it to a MaxHeap

initial heap

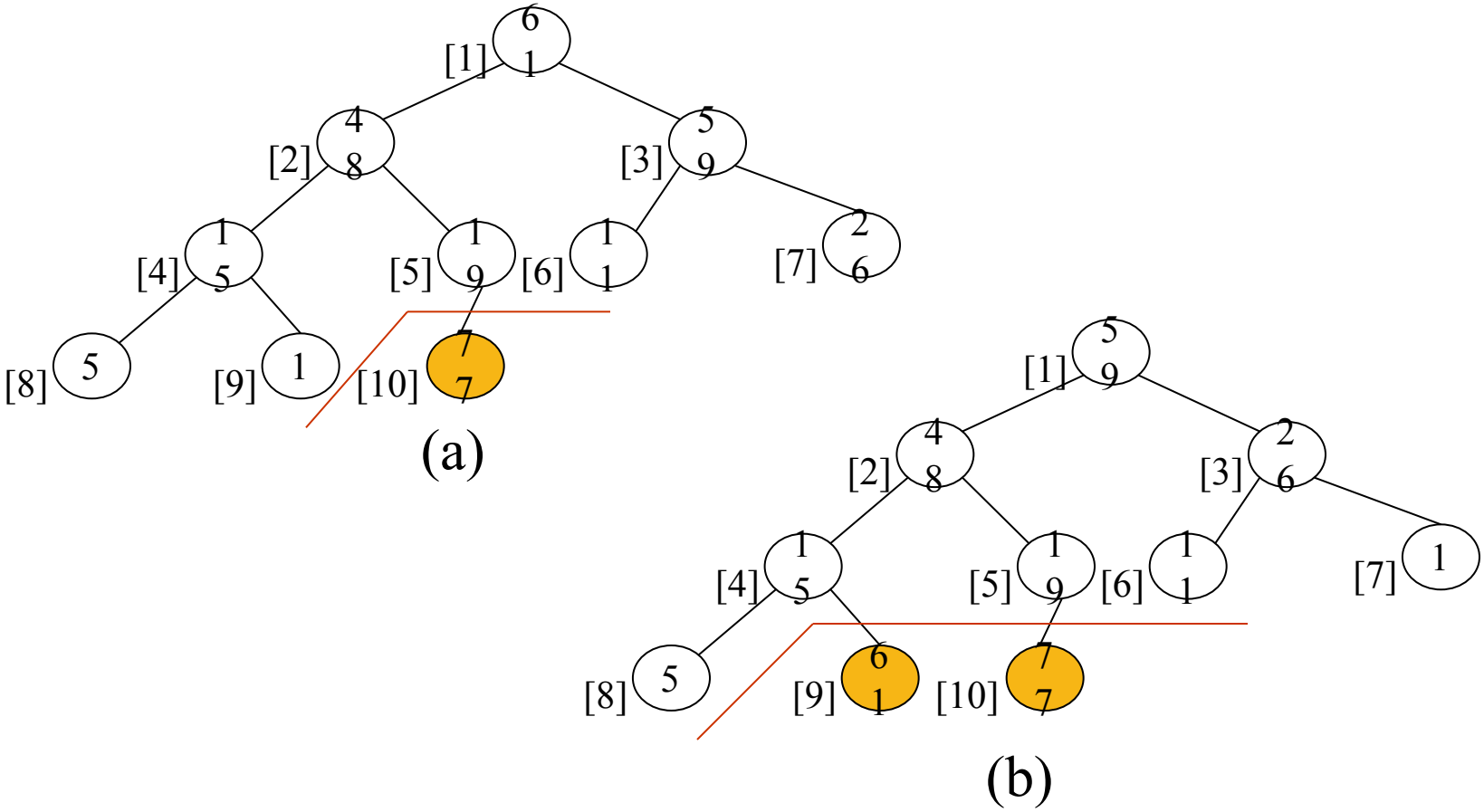


Heap Sort

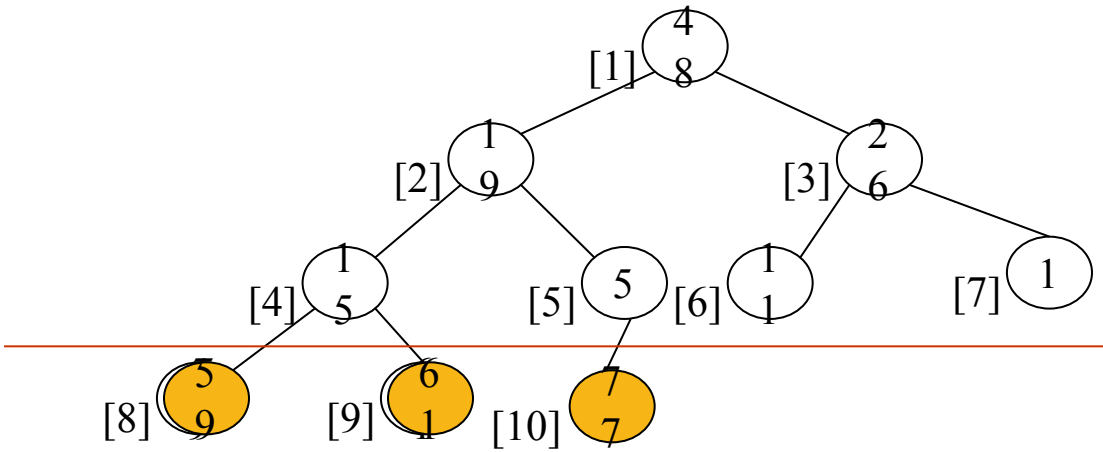
- Exchange and adjust



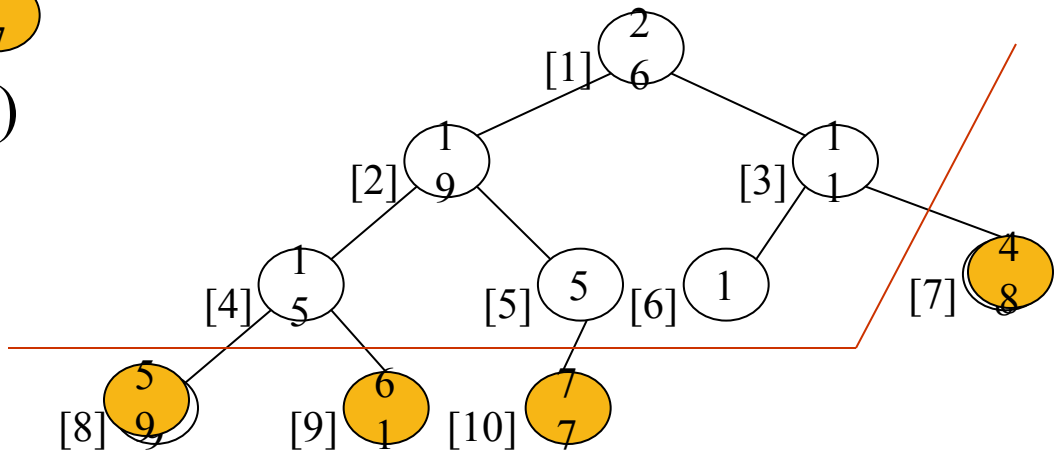
Heap Sort


[1] 6 1
[2] 4 8
[3] 5 9
[4] 1 5
[5] 1 9
[6] 1 1
[7] 2 6
[8] 5
[9] 1
[10] 7 7
(a)
[1] 5 9
[2] 4 8
[3] 2 6
[4] 1 5
[5] 1 9
[6] 1 1
[7] 1
[8] 5
[9] 6 1
[10] 7 7
(b)

Heap Sort

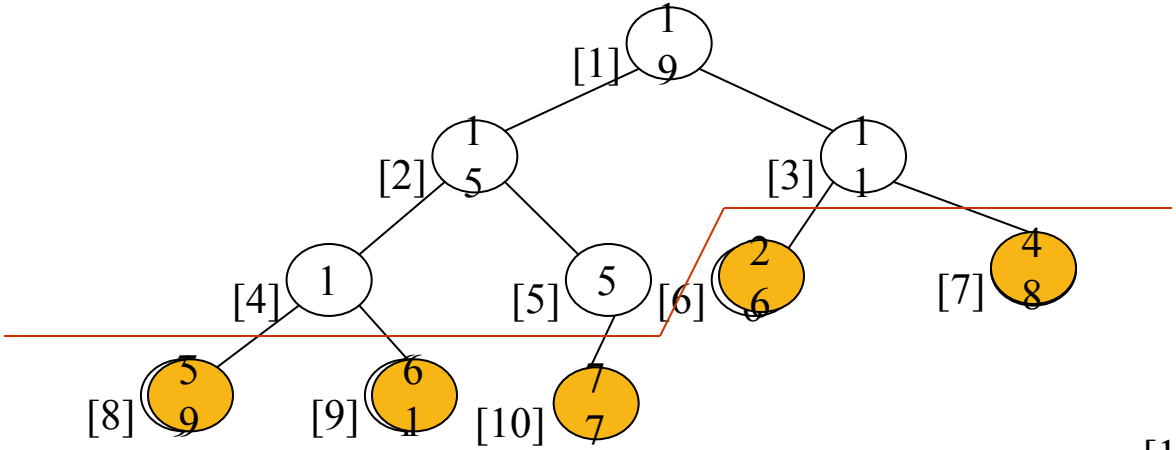

[1] 48
[2] 19
[3] 26
[4] 15
[5] 5
[6] 11
[7] 1
[8] 59
[9] 61
[10] 77

(c)

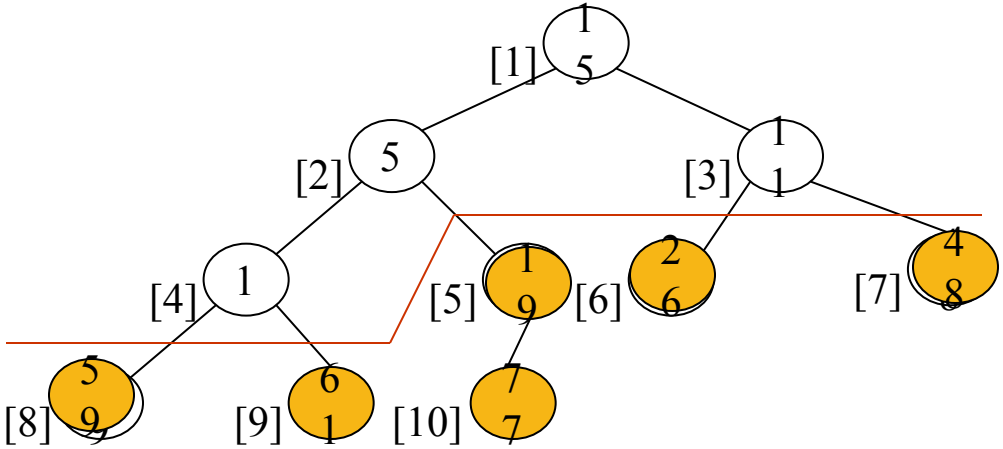

[1] 26
[2] 19
[3] 11
[4] 15
[5] 5
[6] 1
[7] 48
[8] 59
[9] 61
[10] 77

(d)

Heap Sort

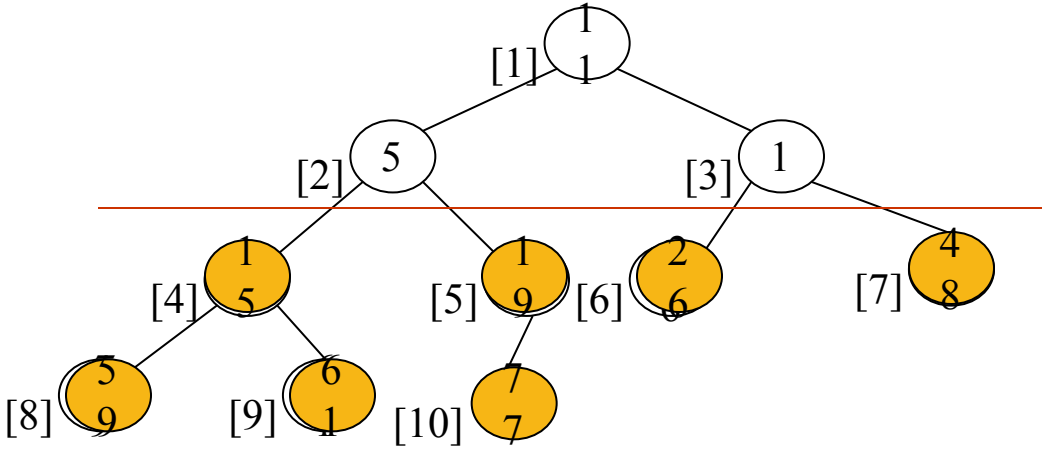

[1] 19
[2] 15
[3] 11
[4] 1
[5] 5
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

(e)

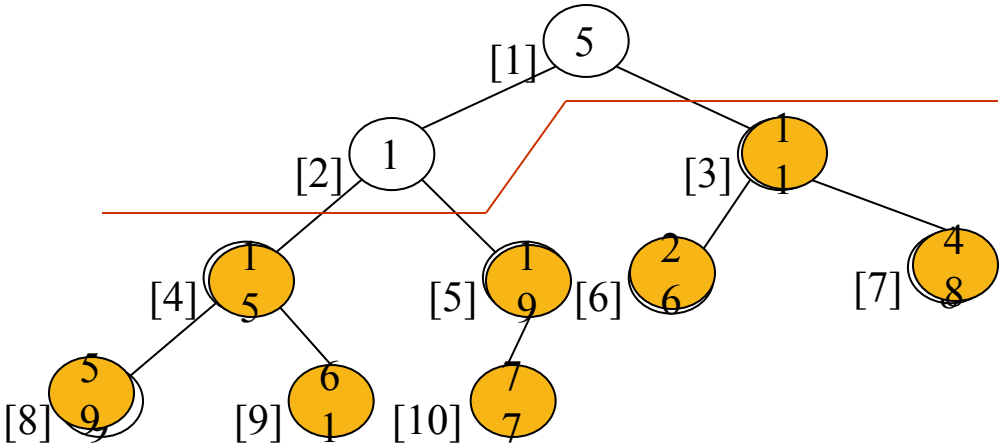

[1] 15
[2] 5
[3] 11
[4] 1
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

(f)

Heap Sort

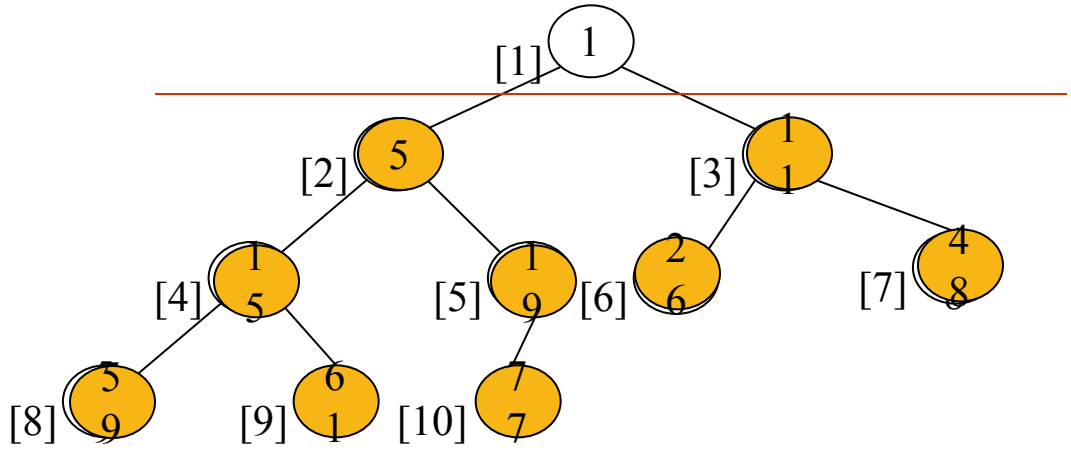

[1]
11
[2]
5
[3]
1
[4]
15
[5]
19
[6]
26
[7]
48
[8]
59
[9]
61
[10]
77

(g)


[1]
5
[2]
1
[3]
11
[4]
15
[5]
19
[6]
26
[7]
48
[8]
59
[9]
61
[10]
77

(h)

Heap Sort



- So results (i)

77 61 59 48 26 19 15 11 5 1